UCRL- 85071
PREPRINT

SUDDENLY PRAXIS

James R. Greenwood

1980 Fall DECUS U.S. Symposium
Town & Country Hotel
San Diego, CA.
November 4-7, 1980

October 22, 1980

Lawrence Livermore Laboratory

SUDDENLY PRAXIS*

James R. Greenwood
Lawrence Livermore National Laboratory
Livermore, CA. 94550

ABSTRACT

Praxis - The practice of the programming art, science, and skill.

Praxis is a high order language designed for efficient programming of control and systems applications. Praxis is a comprehensive, strongly typed, block structured language in the tradition of Pascal, with much of the power of the languages Mesa and Ada. The language supports the development of systems composed of separate modules, user-defined data types, exception handling, detailed control mechanisms, and encapsulated data and routines. Direct access to machine facilities, efficient bit manipulation, and interlocked critical regions are provided within the language.

*Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

INTRODUCTION

Praxis is a modern block structured fully typed algorythmic programming language for control and system implementation applications. It's design has been influenced by the languages. Simula, BCPL, Euclid, PL/I, Jovial, CS-4, Pascal, Alphard, Mesa, and Bliss; as well as by the Department of Defense's language development effort and the proposed Ada language. As to scope and power, Praxis most closely resembles Ada and Mesa.

The control environment differs in important ways from application to application and machine to machine; the language must have features to handle these differences. High level facilities which mask machine dependencies and foster machine independence (portability) usually prevent exactly the programming capability needed for real-time control applications programming. Praxis is a high level language with controlled machine dependent access methods.

The language is "strongly typed". The programmer is given a collection of predefined types and has the ability to construct new types. Every variable, constant, parameter, and expression has a type. All types can be deduced at compile-time and the compiler requires that each value be used in a way consistent with the rules associated with its type. For instance, it is a compile-time error to attempt to pass an integer parameter to a routine which requires a real parameter.

The language is "blocked structured". Blocks are a method of packaging statements and declarations such that the scope of the statements is clearly specified and controlled. The language has more than fifteen block structured statements, each of which is delimited by a form "XXX/endXXX" pair where "XXX" represents the particular statement name.

For instance:

```
for . . . . . . endfor
if  . . . . . . endif
procedure . . . endprocedure
select  . . . . endselect
```

The block structuring also enforces a particular programming style which has been found to be more readable and maintainable than unstructured programming.

INTENDED APPLICATION

The language has been designed for control and communications programming applications. In addition, system programming applications have been found to require the same language facilities.

These applications impose stringent requirements in such areas as:

- efficiency of object code.
- direct access to machine facilities.
- efficient bit manipulation.
- complex data and control structures.
- large programs developed by a team.
- maintenance and upgrades.

Also the programmer of these applications requires detailed control over the code produced by the compiler, in such areas as optimization, variable allocation and implicit run-time support. It is important in these applications that exactly "what is going on" is explicitly represented by the programmer.

PROGRAMMING EXAMPLES

A simple example in the language is the matrix multiply of two N by N matrices named "specA" and "specB" and storing result in "Spectrum".

```
for I := to N do
   for J := to N do
      T := 0
      for K := 1 to N do
         T := T + specA [I,K] * specB [K,J]
      endfor
      Spectrum [I,J] := T
   endfor
endfor
```

This example only makes sense within the scope of the declarations for the variables used. All the variables, except the "for" loop indices must be declared before use. Thus the code above would be preceded by something of the form:

```
declare
   N = 32
   T : integer
   specA : array [1..N,1..N] of integer
   specB : array [1..N,1..N] of integer
   Spectrum : array [1..N,1..N] of integer
enddeclare
```

This declaration block could be written more efficiently in various forms. One method would be to use a user defined type for the array declarations which then would enforce that the three arrays are all the same type, and remain so with subsequent software maintenance. Thus the declarations could take the form:

```
declare
   N = 32
   T : integer
   matrix is array [1..N,1..N] of integer
   specA : matrix
   specB : matrix
   spectrum : matrix
enddeclare
```

Note that we have used the language's comment convention "//" which designates that all text to the right on the line is treated as a comment. In addition this LRM underlines all language reserved words in the examples.

Another example is a simple exchange sort in which an array values is sorted into ascending order:

```
declare
   N = 100
   data : array [1..N] of integer
   done : boolean
enddeclare
... code to store values in data ...
repeat
   done := true
   for K := 2 to N do
      if data [K-1] > data [K] do
         swap (data[K-1], data[K])
         done := false
      endif
   endfor
until done
```

The "repeat" block structured statement is an exception to the ending syntax rule, in that the "until" is the end for the repeat block. The "repeat/until" has the semantics that the included statements are executed repeatedly until such a time that the expression after the "until" is true. Other looping constructs are available in the language including the while/endwhile, and three forms of the for/endfor.

A more detailed control programming application is shown below, which directly reads a hardware input/output device on a PDP-11 computer in a multiprocess environment. In this example the resource (i.e., I/O device) is protected by the interlock variable padlock in a critical "region". Another process with similar code using the same resource cannot preempt the critical region code sequence.

```
Declare
   status : volatile location (8!176420) logical
   datum  : volatile location (8!176422) char
   padlock : static interlock
   temp : char
enddeclare
...
Region padlock do
   Repeat until status and 8#200
   temp := datum
endregion
```

The attribute volatile on the variables "status" and "datum" informs the compiler that the variables must be referenced directly each time they are mentioned in the program, and no optimizations are to be performed on those variables. This attribute allows variables to be used as I/O registers as above, as well as to be used in shared memory.

The location attribute informs the compiler to place the variable in the physical address specified by the octal (8!) constant in the parenthesis. The variable is static and always resides at that location.

A more complex application which demonstrates the ability in the language to bypass the strong typing (when desired) is the function "Upper" which converts a lower-case letter to an uppercase letter.

```
function Upper (inchar:char) returns outchar:char
   declare
      bits is 8 bit logical
      mask = 8#337
      lows is set of char range $a..$z
      lower = lows($a to $z)
   enddeclare
   if inchar in lower do
      outchar := inchar
      return
   endif
   outchar := force char ((force bits (inchar)) and mask)
endfunction {Upper}
```

In the above, the force explicitly overrides the type-checking mechanism, thus allowing a logical operation to be performed on the bit string representing the input character.

LANGUAGE FEATURES

This section outlines some of the major features of Praxis.

Control and Iteration Statements

```
-  while expr do
           sentence; . . . endwhile
-  repeat sentence; . . . until expr
-  if expr do sentence; . . . orif expr do
           sentence; . . . otherwise
           sentence; . . . endif
-  select expr from case item : sentence; . . .
           default : sentence; . . . endselect
-  Upon id, . . . leave
           sentence; . . . through
           case id : sentence; . . . endupon
-  via id
-  break label
-  loop label
-  For id := expr to expr do
           sentence; . . . endfor
-  For id := expr downto expr do
           sentence; . . . endfor
-  For id in expr do sentence; . . . endfor
-  For id := expr then expr while expr do
           sentence; . . . endfor
```

Predefined Types

A type specifies a set of properties and attributes for elements of that type, and the permissable usage of variables of that type. The predefined types are:

integer	real	char	boolean
logical	interlock	cardinal	enumeration
array	structure	function	procedure
pointer	set	long_real	

Most are common to other Pascal-like languages, with the exception of procedure and function variables, and interlocks.

User defined types and abstract data types can be declared. All types can be determined at compile-time and the compiler requires that each usage be consistent with the rules for that type. This includes strong type checking between separately compiled modules.

Routines

Two forms of routines are available; procedures and functions. Formal parameters are typed and are checked for consistency at compile-time with the actual parameters passed on the invocation. Procedure declarations take the form:

```
Procedure Test (X : integer, Y : ref real)
   sentence; . . .
endprocedure {Test}
```

and functions are the form:

```
    Function Zap (Z : blap) returns t : zip
       sentence; . . .
    endfunction {Zap}
```

Explicit exit from routines can be programmed by the return statement.

The formal parameter specification allows the passing of parameters by Ref or Val in any of three modes in, inout, and out. In addition, keyword (i.e., named) formal parameters are available, as well as, optional parameters.

Flexible arrays can be used as formal parameters to allow routines which take any size array as parameters. Fortran and Interrupt routine linkages are implemented.

Modules

A module concept is used to encapsulate data and code. The separate complilation mechanism, abstract data type declaration, and data and code protection are implemented with modules. The form is:

```
    Module Test
       Import id, . . . from Module
       Import id, . . . from Module2
       Export id, . . .
       Export id, . . . to Module3
       Use Module4
           sentence; . . . .
    endmodule {Test }
```

The type attributes hidden and readonly restrict data access on items exported with those attributes.

Modules may be nested and separately compiled. Type checking is done between separately compiled modules. Whole modules can be "imported" by the Use statement. The imports and uses are at any level as normal declarations.

Exception Handling

User defined exceptions and guard blocks allow programming of error and abort conditions. Exceptions are declared with the syntax:

```
Exception reached_limit, overflow
```

and utilized within the block structured statement:

```
Guard sentence; . . . catch
         case reached_limit : sentence; . . .
         default : sentence; . . . endguard
```

Exceptions are dynamically scoped and invoked by the statements:

```
raise reach_limit
reraise
raise overflow finishing lock
```

IMPLEMENTATION STATUS

Three compilers for Praxis are operational and nearly complete:

VAX/VMS generating native code
VAX/VMS generating PDP-11 code
PDP-11/RSX-11M generating PDP-11 code

The compilers written in Praxis are self supporting on each system and have been in use for eight months.

Praxis Compilers under UNIX and RT-11 are being investigated.